

# Width, depth and space

Li-Hsuan Chen                      Felix Reidl  
 clh100p@cs.ccu.edu.tw          fjreidl@ncsu.edu

Peter Rossmanith  
 rossmani@cs.rwth-aachen.de

Fernando Sánchez Villaamil  
 fernando.sanchez@cs.rwth-aachen.de

August 9, 2016

## Abstract

The width measure *treedepth*, also known as vertex ranking, centered coloring and elimination tree height, is a well-established notion which has recently seen a resurgence of interest. Since graphs of bounded *treedepth* are more restricted than graphs of bounded tree- or pathwidth, we are interested in the algorithmic utility of this additional structure.

On the negative side, we show with a novel approach that every dynamic programming algorithm on *treedepth* decompositions of depth  $t$  cannot solve DOMINATING SET with  $O((3 - \epsilon)^t \cdot \log n)$  space for any  $\epsilon > 0$ . This result implies the same space lower bound for dynamic programming algorithms on tree and path decompositions. We supplement this result by showing a space lower bound of  $O((3 - \epsilon)^t \cdot \log n)$  for 3-COLORING and  $O((2 - \epsilon)^t \cdot \log n)$  for VERTEX COVER. This formalizes the common intuition that dynamic programming algorithms on graph decompositions necessarily consume a lot of space and complements known results of the *time*-complexity of problems restricted to low-treewidth classes [16].

We then show that *treedepth* lends itself to the design of branching algorithms. This class of algorithms has in general distinct advantages over dynamic programming algorithms: a) They use less space than algorithms based on dynamic programming, b) they are easy to parallelize and c) they provide possible solutions before terminating, i.e. they can be used to derive heuristics by using preliminary solutions. Specifically, we design two novel algorithms for DOMINATING SET on graphs of *treedepth*  $t$ : a pure branching algorithm that runs in time  $t^{O(t^2)} \cdot n$  and uses space  $O(t^3 \log t + t \log n)$  and a hybrid of branching and dynamic programming that achieves a running time of  $O(3^t \log t \cdot n)$  while using  $O(2^t t \log t + t \log n)$  space. Algorithms for 3-COLORING and VERTEX COVER with space complexity  $O(t \cdot \log n)$  and time complexity  $O(3^t \cdot n)$  and  $O(2^t \cdot n)$ , respectively, are included for completeness.

## 1 Introduction

The notion of *treedepth* has been introduced several times in the literature under several different names. It was first formally studied in the context of matrix

factorization as *minimum elimination trees* [22]; Katchalski *et al.* studied the same notion under the name of *ordered colorings* [14]; Bodlaender *et al.* used the term *vertex ranking* [6]. Recently, Ossona de Mendez and Nešetřil brought the same concept to the limelight in the guise of *treedepth* in their book *Sparsity* [18]. Here we use that definition of treedepth which one we consider easiest to exploit algorithmically: The treedepth  $\mathbf{td}(G)$  of a graph  $G$  is the minimal height of a forest  $F$  such that  $G$  is a subgraph of the closure of  $F$ . The closure of a forest is the graph resulting from adding edges between every node and its ancestors, *i.e.* making every path from root to leaf into a clique. A *treedepth decomposition* is a forest witnessing this fact.

Algorithmically, treedepth is interesting since it is structurally more restrictive than pathwidth. Treedepth bounds the pathwidth and treewidth of a graph, *i.e.*  $\mathbf{tw}(G) \leq \mathbf{pw}(G) \leq \mathbf{td}(G) - 1 \leq \mathbf{tw}(G) \cdot \log n$ , where  $\mathbf{tw}(G)$  and  $\mathbf{pw}(G)$  are the treewidth and pathwidth of an  $n$ -vertex graph  $G$  respectively. Furthermore, a path decomposition can be easily computed from a treedepth decomposition. Not only are there problems that are  $W[1]$ -hard when parameterized by treewidth or pathwidth, but  $\text{fpt}$  when parameterized by treedepth [13]; low treedepth can also be exploited to count the number of appearances of different substructures, such as matchings and small subgraphs, much more efficiently [9, 12].

It turns out, however, that the ‘classical’ approach of dynamic programming (DP) on graph decompositions cannot possibly provide us with faster algorithms, even if we restrict ourselves to treedepth decompositions. Worse, we show in the following that the *space* complexity is necessarily high. To clarify, we consider algorithms that take as input a tree-, path- or treedepth decomposition of width  $s$  and size  $n$  and satisfy the following constraints:

1. They pass a single time over the decomposition in a bottom-up fashion;
2. they use  $O(f(s) \log n)$  space; and
3. they do not modify the decomposition, including re-arranging it.

While these three constraints might look stringent, they include pretty much all dynamic programming algorithm for hard optimization problems on tree or path decompositions. For that reason, we will refer to this type of algorithms simply as *DP algorithms* in the following.

In order to show the aforementioned space lower bounds, we introduce a simple machine model that models DP algorithms on treedepth decompositions and construct superexponentially large Myhill-Nerode families that imply lower bounds for DOMINATING SET, VERTEX COVER, and 3-COLORABILITY in this algorithmic model. These lower bounds hold as well for tree and path decompositions and align nicely with the space complexity of known DP algorithms: for every  $\epsilon > 0$ , no DP algorithm on such decomposition of width/depth  $s$  can use space bounded by  $O((3 - \epsilon)^s \cdot \log n)$  for 3-COLORING or DOMINATING SET nor  $O((2 - \epsilon)^s \cdot \log n)$  for VERTEX COVER. While probably not surprising, we consider a formal proof for what previously were just widely held assumptions valuable. The provided framework should easily extend to other problems.

Consequently, any algorithmic benefit of treedepth over pathwidth and treewidth must be obtained by non-DP means. We demonstrate that treedepth allows the design of branching algorithms whose space consumption grows only polynomially in the treedepth and logarithmic in the input size. Such space-efficient algorithms are quite easy to obtain for 3-COLORING and VERTEX COVER

with running time  $O(3^t \cdot n)$  and  $O(2^t \cdot n)$ , respectively, and space complexity  $O(t \cdot \log n)$ . Our main contribution on the positive side here are two linear-fpt algorithms for DOMINATING SET which use much more involved branching rules on treedepth decompositions. The first one runs in time  $t^{O(t^2)} \cdot n$  and uses space  $O(t^3 \log t + t \cdot \log n)$ . Compared to simple dynamic programming, the space consumption is improved considerably, albeit at the cost of a much higher running time. For this reason, we design a second algorithm that uses a hybrid approach of branching and dynamic programming, resulting in a competitive running time of  $O(3^t \log t \cdot n)$  and space consumption  $O(2^t t \log t + t \log n)$ . Both algorithms are amenable to heuristic improvements (see the Conclusion for a discussion), making them good candidates for real-world application.

While applying branching to treedepth seems natural, it is unclear whether it could be applied to treewidth or pathwidth. Very recent work by Drucker, Nederlof, and Santhanam suggests that, relative to a collapse of the polynomial hierarchy, INDEPENDENT SET restricted to low-pathwidth graphs cannot be solved by a branching algorithm in fpt-time [11].

The idea of using treedepth to improve space consumption is not entirely novel. Fürer and Yu demonstrated that it is possible to count matchings using polynomial space in the size of the input [12] and a parameter closely related to the treedepth of the input. Their algorithm achieves a small memory footprint by using the algebraization framework developed by Loksthanov and Nederlof [17]. This technique was also used to develop an algorithm for DOMINATING SET which runs in time  $3^t \cdot \text{poly}(n)$  (non-linear) and uses space  $O(t \cdot \log n)$  [19]. In our opinion, this type of algorithm has two disadvantages: On the theoretical side, the dependency on the input size is at least  $\Omega(n)$ . A dependence of  $O(\log n)$ , as provided by dynamic programming, would be preferable. On the practical side, using the *Discrete Fourier Transform* makes it hard to apply common algorithm engineering techniques, like *branch & bound*, which are available for branching algorithms.

## 2 Preliminaries

We write  $N_G[x]$  to denote the closed neighbourhood of  $x$  in  $G$  and extend this notation to vertex sets via  $N_G[S] := \bigcup_{x \in S} N_G[x]$ . Otherwise we use standard graph-theoretic notation (see [10] for any undefined terminology). All our graphs are finite and simple and logarithms use base two. For reasons of space we defer some proofs to the appendix and mark the respective claim with a  $\star$ . For sets  $A, B, C$  we write  $A \uplus B = C$  to express that  $A, B$  partition  $C$ .

**Definition 2.1** (Treedepth). *A treedepth decomposition of a graph  $G$  is a forest  $F$  with vertex set  $V(G)$ , such that if  $uv \in E(G)$  then either  $u$  is an ancestor of  $v$  in  $F$  or vice versa. The treedepth  $\text{td}(G)$  of a graph  $G$  is the minimum height of any treedepth decomposition of  $G$ .*

We assume that the input graphs are connected, which allows us to presume that the treedepth decomposition is always a tree. Furthermore, let  $x$  be a node in some treedepth decomposition  $T$ . We denote by  $T_x$  the complete subtree rooted at  $x$  and by  $P_x$  the set of ancestors of  $x$  in  $T$  (not including  $x$ ). A *subtree* of  $x$  refers to a subtree rooted at some child of  $x$ . The treedepth of a graph  $G$  bounds

its *treewidth*  $\mathbf{tw}(G)$  and *pathwidth*  $\mathbf{pw}(G)$ , i.e.  $\mathbf{tw}(G) \leq \mathbf{pw}(G) \leq \mathbf{td}(G) - 1$  [18]. For a definition of treewidth and pathwidth see e.g. Bodlaender [3].

An *s-boundaried graph*  ${}^\circ G$  is a graph  $G$  with a set  $bd({}^\circ G) \subseteq V(G)$  of  $s$  distinguished vertices labeled 1 through  $s$ , called the *boundary* of  ${}^\circ G$ . We will call vertices that are not in  $bd({}^\circ G)$  *internal*. By  $\mathcal{G}_s$  we denote the class of all  $s$ -boundaried graphs. For  $s$ -boundaried graphs  ${}^\circ G_1$  and  ${}^\circ G_2$ , we let the *gluing* operation  ${}^\circ G_1 \oplus {}^\circ G_2$  denote the  $s$ -boundaried graph obtained by first taking the disjoint union of  $G_1$  and  $G_2$  and then unifying the boundary vertices that share the same label<sup>1</sup>.

### 3 Space lower bounds for dynamic programming

Lokshtanov, Marx, and Saurabh showed—assuming SETH—that algorithms for 3-COLORING, VERTEX COVER and DOMINATING SET on a tree decomposition of width  $w$  with running time  $O(3^w \cdot n)$ ,  $O(2^w \cdot n)$  and  $O(3^w w^2 \cdot n)$ , respectively, are basically optimal [16]. Their stated intent (as reflected in the title of the paper) was to substantiate the common belief that known DP algorithms that solve these problems were optimal. This is why we feel that a restriction to a certain type of algorithm is not necessarily inferior to a complexity-based approach: indeed, most algorithms leveraging treewidth *are* dynamic programming algorithms or can be equivalently expressed as such [3, 4, 5, 7, 8, 21]. Even before dynamic programming on tree-decompositions became an important subject in algorithm design, similar concepts were already used implicitly [1]. The sentiment that the table size is the crucial factor in the complexity of dynamic programming algorithms is certainly not new (see e.g. [20]), so it seems natural to provide lower bounds to formalize this intuition. Our tool of choice will be a family of boundaried graphs that are distinct under Myhill-Nerode equivalence. The perspective of viewing graph decompositions as an ‘algebraic’ expression of boundaried graphs that allow such equivalences is well-established [5, 8].

First of all, we need to establish what we mean by *dynamic programming*. DP algorithms on graph decompositions work by visiting the bags/nodes of the decomposition in a bottom-up fashion (a post-order depth-first traversal), maintaining DP tables to compute a solution. For decision problems, these algorithms only need to keep at most  $\log n$  tables in memory at any given moment. We propose a machine model with a read-only tape for the input that can only be traversed once, which only accepts as input decompositions presented in a valid order. This model suffices to capture known dynamic programming algorithms on path, tree and treedepth decompositions. More specifically, given a decision problem on graphs  $\Pi$  and some well-formed instance  $(G, \xi)$  of  $\Pi$  (where  $\xi$  encodes the non-graph part of the input), let  $T$  be a tree, path or treedepth decomposition of  $G$  of width/depth  $k$ . We fix an encoding  $\hat{T}$  of  $T$  that lists the separators provided by the decomposition in the order they are normally visited in a dynamic programming algorithm (post-order depth-first traversal of the bag/nodes of a tree/path/treedepth decomposition) and additionally encodes the edges of  $G$  contained in a separator using  $O(k \log k)$  bits per bag. Then  $(k, \hat{T}, \xi)$  is a well-formed instance of the *DP decision problem*  $\Pi_{DP}$ . Pairing DP

<sup>1</sup>In the literature the result of gluing is often an unboundaried graph. Our definition of gluing will be more convenient in the following.

decision problems with the following machine model provides us with a way to model DP computation over graph decompositions.

**Definition 3.1** (Dynamic programming TM). *A DPTM  $M$  is a Turing machine with an input read-only tape, whose head moves only in one direction, and a separate working tape. It accepts as inputs well-formed instances of some DP decision problem.*

Any single-pass dynamic programming algorithm that solves a DP decision problem on tree, path or treedepth decompositions of width/depth  $k$  using tables of size  $f(k)$  that does not re-arrange the decomposition can be translated into a DPTM with a working tape of size  $O(f(k) \cdot \log n)$ . This model does not suffice to rule out algebraic techniques, since this technique, like branching, requires to visit every part of the decomposition many times [12]; or algorithms that preprocess the decomposition first to find a suitable traversal strategy.

The following notion of a *Myhill-Nerode family* will provide us with the machinery to prove space lower-bounds for DPTMs and hence common dynamic programming algorithms. Recall that  $\mathcal{G}_s$  denotes the class of all  $s$ -boundaried graphs.

**Definition 3.2** (Myhill-Nerode family). *A set  $\mathcal{H} \subseteq \mathcal{G}_s \times \mathbb{N}$  is a  $s$ -Myhill-Nerode family for a DP-decision problem  $\Pi_{DP}$  if the following holds:*

1. *Every  $s$ -boundaried graph  ${}^\circ H \in \mathcal{G}_s$  appears at most once in  $\mathcal{H}$  and if it does appear its size is bounded by  $|{}^\circ H| \leq |\mathcal{H}| \text{polylog}|\mathcal{H}|$ .*
2. *For every subset  $\mathcal{I} \subseteq \mathcal{H}$  there exists an  $s$ -boundaried graph  ${}^\circ G_{\mathcal{I}}$  with  $|{}^\circ G_{\mathcal{I}}| \leq |\mathcal{H}| \text{polylog}|\mathcal{H}|$  and an integer  $p_{\mathcal{I}}$  such that for every  $(H, q) \in \mathcal{H}$  it holds that*

$$({}^\circ G_{\mathcal{I}} \oplus {}^\circ H, p_{\mathcal{I}} + q) \in \Pi_{DP} \iff (H, q) \notin \mathcal{I}.$$

We define the *size* of a Myhill-Nerode family  $\mathcal{H}$  as  $|\mathcal{H}|$  and its *treedepth* as

$$\mathbf{td}(\mathcal{H}) = \max_{({}^\circ H, \cdot) \in \mathcal{H}, \mathcal{I} \subseteq \mathcal{H}} \mathbf{td}({}^\circ G_{\mathcal{I}} \oplus {}^\circ H \oplus {}^\circ K_s).$$

We can similarly define the treewidth and pathwidth of a family. Note that by turning the boundary in the above definition into a clique we essentially measure the treedepth under the assumption that all boundary vertices appear as a path of length  $s$  on the top of the decomposition. The following lemma still holds if we replace ‘treedepth’ by ‘pathwidth’ or ‘treewidth’.

**Lemma 3.3.** *Let  $\epsilon > 0, c > 1$  and  $\Pi$  be a DP decision problem such that for every  $s$  there exists an  $s$ -Myhill-Nerode family  $\mathcal{H}$  for  $\Pi$  of size  $c^s / \text{poly}(s)$  and depth  $\mathbf{td}(\mathcal{H}) = s + o(s)$ . Then no DPTM can decide  $\Pi$  using space  $O((c - \epsilon)^k \log n)$ , where  $n$  is the size and  $k$  the depth of the treedepth decomposition given as input.*

*Proof.* Assume to the contrary that such a DPTM  $M$  exists. Fix  $s$  and consider any subset  $\mathcal{I} \subseteq \mathcal{H}$  of the  $s$ -Myhill-Nerode family  $\mathcal{H}$  of  $\Pi$ . By definition and our above assumptions, all graphs in  $\mathcal{H}$  and the graph  ${}^\circ G_{\mathcal{I}}$  have size at most

$$|\mathcal{H}| \text{polylog}|\mathcal{H}| = c^s \cdot \text{poly}(s).$$

For every  $s$ -boundaried graph  ${}^\circ H$  contained in  $\mathcal{H}$ , there exist treedepth decompositions for  ${}^\circ G_{\mathcal{I}} \oplus {}^\circ H$  of depth at most  $s + o(s)$  such that the boundary vertices  $bd({}^\circ G_{\mathcal{I}})$  appear on a path of length  $s$  on the top of the decomposition. Hence, we can fix a treedepth decomposition  $T_{\mathcal{I}}$  of  $G_{\mathcal{I}}$  with exactly that property and chose a treedepth decomposition of  ${}^\circ G_{\mathcal{I}} \oplus {}^\circ H$  that contains  $T_{\mathcal{I}}$ . Moreover, we chose an encoding of the resulting decomposition that lists the nodes of  $T_{\mathcal{I}}$  first.

There are  $2^{|\mathcal{H}|} = 2^{c^s/\text{poly}(s)}$  choices for  $\mathcal{I}$  but  $M$  only uses  $(c-\epsilon)^{s+o(s)} \cdot \text{poly}(s+o(s)) = c^{s-o(s)}/\text{poly}(s)$  space. Hence by the pigeonhole principle there exist graphs  ${}^\circ G_{\mathcal{I}}, {}^\circ G_{\mathcal{J}}, \mathcal{I} \neq \mathcal{J} \subseteq \mathcal{H}$  for which  $M$  is in the same state and has the same working-tape content after reading the nodes of the respective decompositions  $T_{\mathcal{I}}$  and  $T_{\mathcal{J}}$ . Choose  $({}^\circ H, q) \in \mathcal{I} \triangle \mathcal{J}$ . By definition, we have that

$$({}^\circ G_{\mathcal{I}} \oplus {}^\circ H, p_{\mathcal{I}} + q) \in \Pi \iff ({}^\circ G_{\mathcal{J}} \oplus {}^\circ H, p_{\mathcal{J}} + q) \notin \Pi$$

but  $M$  will either reject or accept both inputs. Contradiction.  $\square$

The following space lower bounds all follow the same basic construction. We define a problem-specific ‘state’ for the vertices of a boundary set  $X$  and construct two boundaried graphs for it: one graph that enforces this state in any (optimal) solution of the respective problem and one graph that ‘tests’ for this state by either rendering the instance unsolvable or increasing the costs of an optimal solution. We begin with the easiest construction: a Myhill-Nerode family for 3-COLORING.

**Theorem 3.4.** *For every  $\epsilon > 0$ , no DPTM solves 3-COLORING on a tree, path or treedepth decomposition of width/depth  $k$  with space bounded by  $O((3-\epsilon)^k \log n)$ .*

*Proof.* For any  $s$  we construct an  $s$ -Myhill-Nerode family  $\mathcal{H}$  as follows. Let  $X$  be the  $s$  boundary vertices of all the boundaried graphs in the following. Then for every three-partition  $\mathcal{X} = \{R, G, B\}$  of  $X$  we add a boundaried graph  ${}^\circ H_{\mathcal{X}}$  to the family  $\mathcal{H}$  by taking a single triangle  $v_R, v_G, v_B$  and connect the vertices  $v_C$  to all vertices in  $X \setminus C$  for  $C \in \{R, G, B\}$ . Since instances of three-coloring do not need any additional parameter, we ignore this part of the construction of  $\mathcal{H}$  and implicitly assume that every graph in  $\mathcal{H}$  is paired with zero.

To construct the graphs  $G_{\mathcal{I}}$  for  $\mathcal{I} \subset \mathcal{H}$ , we will employ the *circuit gadget*  $v_1, v_2, u$  highlighted in Figure 1: note that if  $v_1, v_2$  receive the same color, then necessarily  $u$  must be colored the same. In every other case, the color of  $u$  is arbitrary. Now for every three-partition  $\mathcal{X} = \{R, G, B\}$  of  $X$ , we construct a *testing gadget*  ${}^\circ T_{\mathcal{X}}$  as follows: for  $C \in \{R, G, B\}$ , we arbitrarily pair the vertices

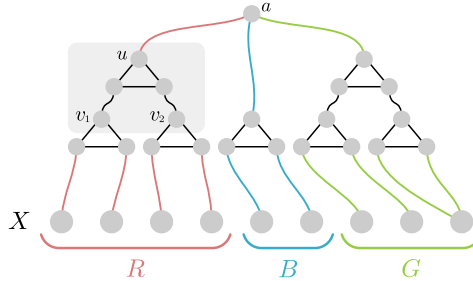


Figure 1: The gadget  ${}^\circ T_{\mathcal{X}}$  for  $\mathcal{X} = \{R, G, B\}$ .

in  $C$  and connect them via the circuit gadget (as  $v_1, v_2$ ). If  $|C|$  is odd, we pair some vertex of  $C$  with itself. We then repeat the construction with all the  $u$ -vertices of those gadgets, resulting in a hierarchical structure of depth  $\sim \log |B_i|$  (cf. Figure 1 for an exemplary construction). Finally, we add a single vertex  $a$  and connect it to the top vertex of the three circuits. Note that by the properties of the circuit gadget, the graph  $\mathring{T}_X$  is three-colorable iff the coloring of  $X$  does *not* induce the partition  $\mathcal{X}$ . In particular, the graph  $\mathring{T}_X \oplus \mathring{H}_{\mathcal{X}'}$  is three-colorable iff  $\mathcal{X} \neq \mathcal{X}'$ .

Now for every subset  $\mathcal{I} \subseteq \mathcal{H}$  of graphs from the family, we define the graph  $\mathring{G}_{\mathcal{I}} = \bigoplus_{H_{\mathcal{X}} \in \mathcal{H}} \Gamma_{\mathcal{X}}$ . By our previous observation, we have that for every  $\mathring{H}_{\mathcal{X}} \in \mathcal{H}$  the graph  $\mathring{G}_{\mathcal{I}} \oplus \mathring{H}_{\mathcal{X}}$  is three-colorable iff  $\mathring{H}_{\mathcal{X}} \notin \mathcal{I}$ . Furthermore, every composite graph has treedepth at most  $s + 4\lceil \log s \rceil + 4$  as witnessed by a decomposition whose top  $s$  vertices are the boundary  $X$ . The graphs  $\mathring{G}_{\mathcal{I}}$  for every  $\mathcal{I} \subseteq \mathcal{H}$  have size at most  $3^s \cdot 6s$ —we conclude that  $\mathcal{H}$  is an  $s$ -Myhill-Nerode family of size  $3^s/6$  (the factor  $1/6$  accounts for the  $3!$  permutations of the partitions) and the claim follows from Lemma 3.3.  $\square$

Surprisingly, the construction to prove a lower bound for VERTEX COVER is very similar to the one for 3-COLORING. For that reason we defer the proof to the appendix.

**Theorem 3.5.** *For every  $\epsilon > 0$ , no DPTM solves VERTEX COVER on a tree, path or treedepth decomposition of width/depth  $k$  with space bounded by  $O((2 - \epsilon)^k \log n)$ .*

*Proof.* Let  $s = |X|$  be the size of the boundary. For every subset  $A \subseteq X$  we construct a graph  $\mathring{H}_A$  for  $\mathcal{H}$  which consists of the boundary and a matching to  $A$  plus  $s - |A|$  isolated  $K_2$ s as padding and add  $(H_A, s)$  to  $\mathcal{H}$ .

Consider  $\mathcal{I} \subseteq \mathcal{H}$ . We will again use the circuit gadget highlighted in Figure 1 to construction  $\mathring{G}_{\mathcal{I}}$ . We assign a budget of two vertices for every such circuit gadget. Note that if one of  $v_1, v_2$  is in the vertex cover, then we can include the top vertex  $u$  into the cover without exceeding this budget. Otherwise,  $u$  cannot be included within the budget. For a set  $A \subseteq X$  we construct the testing gadget  $\mathring{T}_A$  by connecting the vertices of  $X \setminus A$  pairwise via the circuit gadget (using an arbitrary pairing and potentially pairing a leftover vertex with itself). As in the proof of Theorem 3.4, we repeat this construction on the respective  $u$ -vertices of the just added circuits and iterate until we have added a single circuit on the very top. Let us denote the topmost  $u$ -vertex in this construction by  $u'$ . Let  $\lambda$  be the number of circuits added in this fashion, then the total budget for  $\mathring{T}_A$  is  $2\lambda$ . Note that if a vertex of  $X \setminus A$  is inside a vertex cover, then the gadget  $\mathring{T}_A$  has a vertex cover of size  $2\lambda$  that includes its top vertex  $u'$ . Otherwise, no vertex cover of  $\mathring{T}_A$  of size  $2\lambda$  includes  $u'$ .

We construct  $\mathring{G}_{\mathcal{I}}$  by taking  $\bigoplus_{H_A \in \mathcal{I}} \mathring{T}_A$  and adding a single vertex  $a$  that connects to all  $u'$ -vertices of the gadgets  $\{\mathring{T}_A\}_{H_A \in \mathcal{I}}$ . For simplicity, we pad out the graph with isolated  $K_2$ s to ensure that every graph  $\mathring{G}_{\mathcal{I}}, \mathcal{I} \subseteq \mathcal{H}$  constructed in this way has a minimum vertex cover of the same size  $\ell$ .

We claim that  $\mathring{G}_{\mathcal{I}} \oplus \mathring{H}_A$  has a vertex cover of size  $\ell$  iff  $\mathring{H}_A \notin \mathcal{I}$ . If  $H_A \notin \mathcal{I}$ , then every gadget  $\Gamma_{A'}$  that comprises  $\mathring{G}_{\mathcal{I}}$  has  $A' \neq A$ . By the above observation, every gadget  $\Gamma_{A'}$  can therefore be covered with  $2\lambda$  vertices including the vertex  $u'$ ; thus all edges incident to the vertex  $a$  are covered and we stay within budget. If  $H_A \in \mathcal{I}$  then  $\mathring{G}_{\mathcal{I}}$  contains the gadget  $\mathring{T}_A$  and as observed we cannot include its

$u'$ -vertex into the budgeted  $2\lambda$  vertices. Hence, the edge  $u'a$  cannot be covered within the total budget  $\ell$  and we need to invest  $\ell + 1$  vertices to obtain a vertex cover. Letting  $p_{\mathcal{I}} = \ell - s$ , we conclude that  $\mathcal{H}$  is a  $s$ -Myhill-Nerode family of size  $2^s$  and depth  $\text{td}(\mathcal{H}) = s + o(s)$ .  $\square$

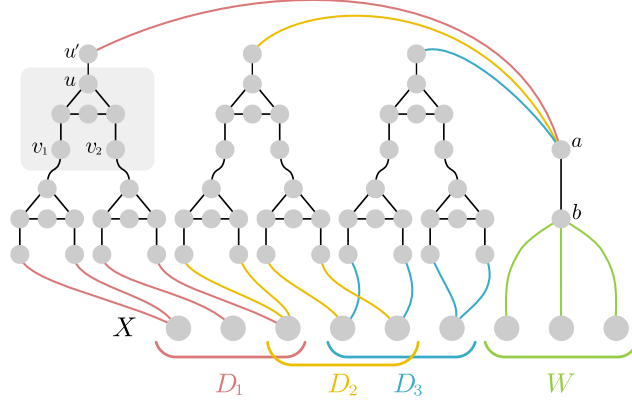


Figure 2: The gadget  $\circ T_W$  for  $\mathcal{D}_W = \{S_1, S_2, S_3\}$ . Padding-vertices are not included.

**Theorem 3.6.** *For every  $\epsilon > 0$ , no DPTM solves DOMINATING SET on a tree, path or treedepth decomposition of width/depth  $k$  with space bounded by  $O((3 - \epsilon)^k \log n)$ .*

*Proof.* For any  $s$  dividable by three we construct an  $s$ -Myhill-Nerode family  $\mathcal{H}$  as follows. Let  $X$  be the  $s$  boundary vertices of all the boundaried graphs in the following. Then for every three-partition  $\mathcal{X} = (B, D, W)$  of  $X$  into sets of size  $s/3$ , we construct a graph  $H_{\mathcal{X}}$  by connecting two pendant vertices to every vertex in  $B$ , connecting every vertex in  $D$  to a vertex with two pendant vertices and leaving  $W$  untouched. Intuitively, we want the vertices of  $B$  to be in any minimal dominating set, the vertices in  $D$  to be dominated from a vertex in  $H_{\mathcal{X}}$  and the vertices in  $W$  to be dominated from elsewhere. We add every pair  $(H_{\mathcal{X}}, s/3 + 1)$  to  $\mathcal{H}$ .

For a subset  $\mathcal{I} \subseteq \mathcal{H}$  let  $\mathcal{D}_W = \{D \mid H_{X \setminus (D \cup W), D, W} \in \mathcal{I}_W\}$  be all  $D$ -sets that appear together with a fixed  $W \subseteq X$  in  $\mathcal{I}$ . We construct the graph  $\circ G_{\mathcal{I}}$  using the *circuit gadget*  $v_1, v_2, u$  highlighted in Figure 2: note that if  $v_1, v_2$  are not dominated, then there is no dominating set of the gadget of size two that contains  $u$ . If one of  $v_1, v_2$  is dominated then such a dominating set containing  $u$  exists. For every  $W \subseteq X$  with  $|W| = s/3$  construct a *testing gadget*  $\circ T_W$  using the circuit gadgets as follows. If  $\mathcal{D}_W$  is non-empty we construct  $\circ T_W$  as follows. For every set  $D \in \mathcal{D}_W$  we construct the gadget  $\circ \Lambda_D$  by arbitrarily pairing the vertices in  $D$  and connecting them via the circuit gadget. If  $|D|$  is odd, we pair some vertex of  $D$  with itself. We then repeat the construction with all the  $u$ -vertices of those gadgets, resulting in a hierarchical structure of depth  $\sim \log |D|$ . To finalize the construction of  $\circ \Lambda_D$  we take the  $u$ -vertex of the last layer and connect it to a vertex  $u'$ . Let in the following  $\lambda$  be the number of circuits we used to construct such a  $\circ \Lambda_D$ -gadget (this quantity only depends on  $s$ ). This concludes the construction of  $\circ \Lambda_D$ . If  $\mathcal{D}_W$  is empty, then we let  $\circ T_W$  be a  $K_2$  with one vertex connected to all vertices in  $W$  plus  $\binom{2s/3}{s/3} 2\lambda$  isolated



padding-vertices. Otherwise we obtain  $\circ\Gamma_W$  by taking the graph  $\bigoplus_{D \in \mathcal{D}} \circ\Lambda_D$  and adding two additional vertices  $a, b$  as well as  $((\binom{2s/3}{s/3}) - |\mathcal{D}_W|)2\lambda$  isolated vertices for padding. The vertex  $a$  is connected to all  $u'$  vertices of all the gadgets  $\{\circ\Lambda_D\}_{D \in \mathcal{D}_W}$  and the vertex  $b$  is connected to  $\{a\} \cup W$  (cf. Figure 2 for an exemplary construction).

We assign the budget  $\alpha = \binom{2s/3}{s/3}2\lambda + 1$  for the gadget  $\circ\Gamma_W$ . If  $\mathcal{D}_W = \emptyset$  this budget is sufficient to include all padding-vertices and the endpoint of the  $K_2$  that connects to  $W$ . Hence, we can dominate  $W$  as well as  $\circ\Gamma_W$  within this budget. For non-empty  $\mathcal{D}_W$ , after accounting for the padding-vertices, we are left with a budget of  $2\lambda$  for each gadget  $\circ\Lambda_D$ ,  $D \in \mathcal{D}_W$  that comprise  $\circ\Gamma_W$ . While at least  $2\lambda$  vertices will always be necessary to dominate  $\circ\Lambda_D$ , only if at least one vertex of  $D$  is contained in the dominating set we can dominate the  $u'$ -vertex of  $\Lambda_D$  within budget. In the case that no vertex of  $D$  is in the dominating set  $2\lambda$  vertices only suffice to dominate all vertices of  $\circ\Lambda_D$  except  $u'$ . Hence, the gadget  $\circ\Gamma_W$  can dominate itself *and*  $W$  within the budget  $\alpha$  exactly when for all  $D \in \mathcal{D}_W$ , least one vertex of  $D$  is in the dominating set. Otherwise, we need to include  $a$  into the dominating set and hence our budget does not suffice to include  $b$  in order to dominate  $W$ . Finally we define for every  $\mathcal{I} \subseteq \mathcal{H}$  the graph  $\circ G_{\mathcal{I}} = \bigoplus_{W \subset X, |W|=s/3} \circ\Gamma_W$ .

Let us now show that our boundaried graphs work as intended and calculate the appropriate parameters  $p_{\mathcal{I}}$ . Consider any graph  $\circ H_{B_0, D_0, W_0} \in \mathcal{H}$  where again  $B_0, D_0, W_0$  is a partition of  $X$  into sets of size  $s/3$  and the graph  $\circ G_{\mathcal{I}}$  for any  $\mathcal{I} \subseteq \mathcal{H}$ . We show that  $\circ H_{B_0, D_0, W_0} \oplus G_{\mathcal{I}}$  has a dominating set of size  $\binom{s}{s/3}\alpha + s/3 + 1$  iff  $\circ H_{B_0, D_0, W_0} \notin \mathcal{I}$  (otherwise a minimal dominating set will exceed his budget by one).

We use the sets  $\mathcal{D}_W, W \subseteq X$  as defined previously. First, assume that  $\mathcal{D}_{W_0} = \emptyset$ , that is, for every set  $B, D$  we have that  $H_{B, D, W_0} \notin \mathcal{I}$  and in particular  $H_{B_0, D_0, W_0} \notin \mathcal{I}$ . As observed above, the gadget  $\Gamma_{W_0}$  can dominate itself and  $W_0$  with a budget of  $\alpha$ . All other gadgets  $\Gamma_W, W \neq W_0$  do not need to dominate their respective  $W$ -sets and can therefore include their  $a$ -vertices, accordingly they can all dominate their internal vertices within the assigned budget  $\alpha$ . Including the  $s/3$  vertices of  $B_0$  as well as the one vertex in  $H_{B_0, D_0, W_0}$  used to dominate  $D_0$  this correctly sums up to a dominating set of size  $\binom{s}{s/3}\alpha + s/3 + 1$ . Next, assume that  $\mathcal{D}_{W_0} \neq \emptyset$  and  $D_0 \notin \mathcal{D}_{W_0}$ , i.e. again  $H_{B_0, D_0, W_0} \notin \mathcal{I}$ . Therefore, for every set  $D \in \mathcal{D}_{W_0}$  we have that  $D \cap B_0 \neq \emptyset$ . Since we include  $B_0$  in our dominating set, the gadgets  $\{\circ\Lambda_D\}_{D \in \mathcal{D}_{W_0}}$  that comprise  $\circ\Gamma_{W_0}$  can all dominate their respective  $u'$ -vertices within the assigned budget of  $2\lambda$ . Hence,  $\circ\Gamma_{W_0}$  can dominate  $W_0$  using the assigned budget of  $\alpha$ . All other gadgets  $\Gamma_W, W \neq W_0$  again stay within budget as observed above and we obtain in total a dominating set of size  $\binom{s}{s/3}\alpha + s/3 + 1$ . Finally, consider the case that  $D_0 \in \mathcal{D}_{W_0}$ , i.e.  $H_{B_0, D_0, W_0} \in \mathcal{I}$ . Then the gadget  $\Lambda_{D_0}$  contained in  $\Gamma_{W_0}$  cannot dominate its  $u'$  vertex within a budget of  $2\gamma$ , hence we need to include either  $u'$  or  $a$ . This depletes the budget of  $\Gamma_{W_0}$  and hence we need in total  $\binom{s}{s/3}\alpha + s/3 + 2$  vertices to dominate  $\circ H_{B_0, D_0, W_0} \oplus G_{\mathcal{I}}$ .

Choosing  $p_{\mathcal{I}} = \binom{s}{s/3}\alpha$  completes the construction of  $(\circ\Gamma_{\mathcal{I}}, p_{\mathcal{I}})$  and we conclude that  $\mathcal{H}$  is an  $s$ -Myhill-Nerode family of size  $\Omega(3^s/s)$ . For  $s$  indivisible by three we take the next smaller integer  $s'$  divisible by three and use the  $s'$ -family as the  $s$ -family. It is easy to confirm that the depth of  $\mathcal{H}$  is  $\mathbf{td}(\mathcal{H}) = s + o(s)$  and

the theorem follows from Lemma 3.3.  $\square$

## 4 Dominating Set using $O(t^3 \log t + t \log n)$ space

That branching might be a viable algorithmic design strategy for low-treewidth graphs can easily be demonstrated for problems like 3-COLORING and VERTEX COVER: we simply branch on the topmost vertex of the decomposition and recurse into (annotated) subinstances. For  $c$ -COLORING, this leads to an algorithm with running time  $O(c^t \cdot n)$  and space complexity  $O(t \log n)$ . Since it is possible to perform a depth-first traversal of a given tree using only  $O(\log n)$  space [15], the space consumption of this algorithm can be easily improved to  $O(t + \log n)$ . Similarly, branching solves VERTEX COVER in time  $O(2^t \cdot n)$  and space  $O(t \log n)$ .

The task of designing a similar algorithm for DOMINATING SET is much more involved. Imagine branching on the topmost vertex of the decomposition: while the branch that includes the vertex into the dominating set produces a straightforward recurrence into annotated instances, the branch that excludes it from the dominating set needs to decide *how* that vertex should be dominated. The algorithm we present here proceeds as follows: We first guess whether the current node  $x$  is in the dominating set or not. Recall that  $P_x$  denotes the nodes of the decomposition that lie on the unique path from  $x$  to the root of the decomposition (and  $x \notin P_x$ ). We iterate over every possible partition  $S_1 \uplus \dots \uplus S_l = P_x \cup \{x\}$  into  $l \leq t$  sets of  $P_x \cup \{x\}$ . The semantic of a block  $S_i$  is that we want every element  $S_i$  to be dominated exclusively by nodes from a specific subtree of  $x$ . A recursive call on a child  $y$  of  $x$ , together with an element of the partition  $S_i$ , will return the size of a dominating set which dominates  $V(T_y) \cup S_i$ . The remaining issue is how these specific solutions for the subtrees of  $x$  can be combined into a solution in a space-efficient manner. To that end, we first compute the size of a dominating set for  $T_y$  itself and use this as baseline cost for a subtree  $T_y$ . For a block  $S_i$  of a partition of  $P_x$ , we can now compare the cost of dominating  $V(T_y) \cup S_i$  against this baseline to obtain overhead cost of dominating  $S_i$  using vertices from  $T_y$ . Collecting these overhead costs in a table for subtrees of  $x$  and the current partition, we are able to apply certain reduction rules on these tables to reduce their size to at most  $t^2$  entries. Recursively choosing the best partition then yields the solution size using only polynomial space in  $t$  and logarithmic in  $n$ . Formally, we prove the following:

**Theorem 4.1.** *Given a graph  $G$  and a treedepth decomposition  $T$  of  $G$ , Algorithm 1 finds the size of a minimum dominating set of  $G$  in time  $t^{O(t^2)} \cdot n$  using  $O(t^3 \log t + t \log n)$  bits.*

We split the proof of Theorem 4.1 into lemmas for correctness, running time and used space.

**Lemma 4.2.** *Algorithm 1 called on a graph  $G$ , a treedepth decomposition  $T$  of  $G$ , the root  $r$  of  $T$  and  $P = D = \emptyset$  returns the size of a minimum dominating set of  $G$ .*

*Proof.* If we look at a minimal dominating set  $S$  of  $G$  we can charge every node in  $V(G) \setminus S$  to a node from  $S$  that dominates it. We are thus allowed to treat any node in  $G$  as if it was dominated by a single node of  $S$ . We will prove this lemma by induction, the inductive hypothesis being that a call on a node  $x$  with

arguments  $D = S \cap P_x$  and  $P \subseteq P_x$  being the set of nodes dominated from nodes in  $T_x$  returns  $|S \cap V(T_x)|$ .

It is clear that the algorithm will call itself until a leaf is reached. Let  $x$  be a leaf of  $T$  on which the function was called. We first check the condition at line 2, which is true if either  $x$  is not dominated by a node in  $D$  or if some node in  $P$  is not yet dominated. In this case we have no choice but to add  $x$  to the dominating set. Three things can happen:  $P$  is not fully dominated, which means that it was not possible under these conditions to dominate  $P$ , in which case we correctly return  $\infty$ , signifying that there is no valid solution. Otherwise we can assume  $P$  is dominated and we return 1 if we had to take  $x$  and 0 if we did not need to do so. Thus the leaf case is correct.

We assume now  $x$  is not a leaf and thus we reach line 3. We first add  $x$  to  $P$ , since it can only be dominated either from a node in  $D$  or a node in  $T_x$ . Nodes in  $T_x$  can only be dominated by nodes from  $V(T_x) \cup P_x$ . We assume by induction that  $D = S \cap P_x$  and that  $P$  only contains nodes which are either in  $S$  or dominated from nodes in  $T_x$ . Algorithm 1 executes the same computations for  $D$  and  $D \cup \{x\}$ , representing not taking and taking  $x$  into the dominating set respectively. We must show that the set  $P$  for the recursive calls is correct. There exists a partition of the nodes of  $P$  not dominated by  $D$  (respectively  $D \cup \{x\}$ ) such that the nodes of every element of the partition are dominated from a single subtree  $T_y$  where  $y$  is a child of  $x$ . The algorithm will eventually find this partition on line 5. The baseline value, i.e. the size of a dominating set of  $T_y$  given that the nodes in  $D$  (respectively  $D \cup \{x\}$ ) are in the dominating set, gives a lower bound for any solution. In the lists in  $L$  and  $L'$  we keep the extra cost incurred by a subtree  $T_y$  if it has to dominate an element of the partition. We only need to keep the best  $t$  values for every  $S_i$ : Assume that it is optimal to dominate  $S_i$  from  $T_y$  and there are  $t + 1$  subtrees induced on children of  $x$  whose extra cost over the baseline to dominate  $S_i$  is strictly smaller than the extra cost for  $T_y$ . At least one of these subtrees  $T_{y'}$  is not being used to dominate an element of the partition. This means we could improve the solution by letting  $T_y$  dominate itself and taking the solution of  $T_{y'}$  that also dominates  $S_i$ . Keeping  $t$  values for every element in the partition suffices to find a minimal solution, which is what *find\_min\_solution* does. Since with lines 20 and 21 we take the minimum over all possible partitions and taking  $x$  into the dominating set or not, we get that by inductive assumption the algorithm returns the correct value. The lemma follows since the first call to the algorithm with  $D = P = \emptyset$  is obviously correct.  $\square$

**Lemma 4.3.** *Algorithm 1 runs in time  $t^{O(t^2)} \cdot n$ .*

*Proof.* The running time when  $x$  is a leaf is bounded by  $O(t^2)$ , since all operations exclusively involve some subset of the  $t$  nodes in  $P_x \cup \{x\}$ . Since  $|P| \leq t$  the number of partitions of  $P$  is bounded by  $t^t$ . When  $x$  is not a leaf the only time spent on computations which are not recursive calls of the algorithm are all trivially bounded by  $O(t)$ , except the time spent on *find\_min\_solution*, which can be solved via a matching problem in time  $\text{poly}(t)$ . The number of recursive calls that a single call on a node  $x$  makes on a child  $y$  is  $O(t \cdot t^t)$  which bounds total number of calls on a single node by  $t^{O(t^2)}$ . This proves the claim.  $\square$

**Lemma 4.4.** *Algorithm 1 uses  $O(t^3 \log t + t \log n)$  bits of space.*

---

**Algorithm 1: domset**

---

**Input:** A graph  $G$  and a treedepth decomposition  $T$  of  $G$ , a node  $x$  of  $T$  and sets  $P, D \subseteq V(G)$ .

**Output:** The size of a minimum Dominating Set.

```
1 if  $x$  is a leaf in  $T$  then
2   if  $x \notin N_G[D]$  or  $P \not\subseteq N_G[D]$  then  $D := D \cup \{x\}$  if  $P \not\subseteq N_G[D]$  then return  $\infty$ 
   else if  $x \in D$  then return 1 else return 0
3  $result := \infty$ ;
4  $P := P \cup \{x\}$ ;
5 foreach partition  $S_1 \uplus \dots \uplus S_l$  of  $P$  do
6    $L := |P|$ -element array of ordered lists;
7    $L' := |P|$ -element array of ordered lists;
8    $baseline := 0$ ;
9    $baseline' := 0$ ;
10  foreach child  $y$  of  $x$  in  $T$  do
11     $b := domset(G, T, y, \emptyset, D)$ ;
12     $baseline := baseline + b$ ;
13     $b' := domset(G, T, y, \emptyset, D \cup \{x\})$ ;
14     $baseline' := baseline + b'$ ;
15    for  $S_i \in \{S_1, \dots, S_l\}$  do
16       $c := domset(G, T, y, S_i, D) - b$ ;
17       $c' := domset(G, T, y, S_i, D \cup \{x\}) - b'$ ;
18      Insert  $(c, y)$  into ordered list  $L[i]$  and keep only smallest  $l$  elements;
19      Insert  $(c', y)$  into ordered list  $L'[i]$  and keep only smallest  $l$  elements;
20  /* Find minimal cost of dominating  $\{S_1, \dots, S_l\}$  from  $L$  and  $L'$  by
   e.g. solving appropriate matching problems. */
21   $result := \min(result, find\_min\_solution(L) + baseline)$ ;
22   $result := \min(result, find\_min\_solution(L') + baseline' + 1)$ ;
23 return  $result$ ;
```

---

*Proof.* There are at most  $t$  recursive calls on the stack at any point. We will show that the space used by one is bounded by  $O(t^2 \log t + \log n)$ . Each call uses  $O(t)$  sets, all of which have size at most  $t$ . The elements contained in these sets can be represented by their position in the path to the root of  $T$ , thus they use at most  $O(t^2 \log t)$  space. The arrays of ordered lists  $L, L'$  contain at most  $t^2$  elements and all entries are  $\leq t$  or  $\infty$ : If the additional cost (compared to the baseline cost) of dominating a block  $S_i$  of the current partition from some subtree  $T_y$  exceeds  $|S_i| \leq t$ , we can disregard this possibility—it would be cheaper to just take all vertices in  $S_i$ , a possibility explored in a different branch.

To find a minimal solution from the table we need to avoid using the same subtree to dominate more than one element of the partition; however, at any given moment we only need to distinguish at most  $t^2$  subtrees. Thus the size of the arrays  $L$  and  $L'$  is bounded by  $O(t^2 \log t)$ . The only other space consumption is caused by a constant number of variables ( $result, baseline, baseline', b, b'$  and  $x$ ) all of them  $\leq n$ . Thus the space consumption of a single call is bounded by  $O(t^2 \log t + \log n)$  and the lemma follows.  $\square$

#### 4.1 Fast Dominating Set using $O(2^t t \log t + t \log n)$ space

We have seen that it is possible to solve DOMINATING SET on low-treedepth graphs in a space-efficient manner. However, we traded exponential space against superexponential running time and it is natural to ask whether there is some

middle ground. We present Algorithm 2 to answer this question: its running time  $O(3^t \log t \cdot n)$  is competitive with the default dynamic programming but its space complexity  $O(2^t \log t + t \log n)$  is exponentially better. The basic idea is to again branch from the top deciding if the current node  $x$  is in the dominating set or not. Intertwined in this branching we compute a function which for a subtree  $T_x$  and a set  $S \subseteq P_x$  gives the cost of dominating  $V(T_x) \cup S$  from  $T_x$ . For each recursive call on a node we only need this function for subsets of  $P_x$  which are not dominated. If  $d$  is the number of nodes of  $P_x$  that are currently in the dominating set, the function only needs to be computed for  $2^{t-d}$  sets. This allows us to keep the running time of  $O^*(3^t)$ , since  $\sum_{i=0}^t \binom{t}{i} \cdot 2^{t-i} = 3^t$ , while only creating tables with at most  $O(2^t)$  entries. By representing all values in these tables as  $\leq t$  offsets from a base value, the space bound  $O(2^t \log t + t \log n)$  follows. For two functions  $M_1, M_2$  with domain  $2^U$  for some ground-set  $U$  (realized via associative arrays in the algorithm) we use the notation  $M_1 * M_2$  to denote the convolution  $(M_1 * M_2)[X] := \min_{A \uplus B = X} M_1[A] + M_2[B]$ , for all  $X \subseteq U$ .

---

**Algorithm 2:** domset

---

**Input:** A graph  $G$  and a treedepth decomposition  $T$  of  $G$ , a node  $x$  of  $T$  and a set  $D \subseteq V(G)$ .  
**Output:** The size of a minimum Dominating Set.

```

1  $M, M_1, M_2 :=$  are empty associative arrays. If a set is not in the array its value is  $\infty$ ;
2 if  $x$  is a leaf in  $T$  then
3    $M[N_G[x] \setminus D] := 1$ ;
4   if  $x \in N_G[D]$  then  $M[\emptyset] := 0$  return  $M$ ;
   /* Assume the children of  $x$  are  $\{y_1, \dots, y_l\}$ . */
5 for  $i \in \{1, \dots, l\}$  do
6    $M' := \text{domset}(G, T, y_i, D)$ ;
7    $M_1 := M_1 * M'$ ;
   /*  $x$  is not in the dominating set. Discard entries where  $x$  is undominated. */
8 if  $x \notin N_G[D]$  then delete all entries  $S$  from  $M$  where  $x \notin S$  for  $i \in \{1, \dots, l\}$  do
9    $M' := \text{domset}(G, T, y_i, D \cup \{x\})$ ;
10   $M_2 := M_2 * M'$ ;
11 foreach  $S \in M_2$  do  $M[S] := M[S] + 1$   $M := M_1 * M_2$ ;
   /* Forget  $x$ . */
12 foreach  $S \in M$  where  $x \notin S$  do  $M[S] := \min(M[S], M[S \cup \{x\}])$  Delete all entries  $S$ 
   from  $M$  where  $x \in S$ ;
13 if  $x$  is the root of  $T$  then return  $M[\emptyset]$  else return  $M$ 
```

---

**Theorem 4.5.** For a graph  $G$  with treedepth decomposition  $T$ , Algorithm 2 finds the size of a minimum dominating set in time  $O(3^t \log t \cdot n)$  using  $O(2^t \log t + t \log n)$  bits of space.

We divide the proof into lemmas as before.

**Lemma 4.6.** Algorithm 2 called on  $G, T, r, \emptyset$ , where  $T$  is a treedepth decomposition of  $G$  with root  $r$ , returns the size of a minimum dominating set of  $G$ .

*Proof.* Notice that the associative array  $M$  represents a function which maps subsets of  $P_x \setminus D$  to integers and  $\infty$ . At the end of any recursive call,  $M[S]$  for  $S \subseteq P_x \setminus D$  should be the size of a minimal dominating set in  $T_x$  which

dominates  $T_x$  and  $S$  assuming that the nodes in  $D$  are part of the dominating set. We will prove this inductively.

Assume  $x$  is a leaf. We can always take  $x$  into the dominating set at cost one. In case  $x$  is already dominated we have the option of not taking it, dominating nothing at zero cost. This is exactly what is computed in lines 2–4.

Assume now that  $x$  is an internal, non-root node of  $T$ . First, in lines 5–8 we assume that  $x$  is not in the dominating set. By inductive assumption calling *domset* on a child  $y$  of  $x$  returns a table which contains the cost of dominating  $T_y$  and some set  $S \subseteq P_y \setminus D$ . By merging them all together  $M_1$  represents a function which gives the cost of dominating some set  $S \subseteq (P_x \cup \{x\}) \setminus D$  and all subtrees rooted at children of  $x$ . We just need to take care that  $x$  is dominated. If  $x$  is not dominated by a node in  $D$ , then it must be dominated from one of the subtrees. Thus we are only allowed to retain solutions which dominate  $x$  from the subtrees. We take care of this on line 8. After this  $M_1$  represents a function which gives the cost of dominating some set  $S \subseteq (P_x \cup \{x\}) \setminus D$  and  $T_x$  assuming  $x$  is not in the dominating set. Then we compute a solution assuming  $x$  is in the dominating set in lines 8–11. We first merge the results on calls to the children of  $x$  via convolution. Since we took  $x$  into the dominating set we increase the cost of all entries by one. After this  $M_2$  represents the function which gives the cost of dominating some set  $S \subseteq P_x \setminus D$  and  $T_x$  assuming  $x$  is in the dominating set. We finally merge  $M_1$  and  $M_2$  together with the min-sum convolution. Since we have taken care that all solutions represented by entries in  $M$  dominate  $x$  we can remove all information about  $x$ . We do this in lines 12–12. Finally,  $M$  represents the desired function and we return it. When  $x$  is the root, instead of returning the table we return the value for the only entry in  $M$ , which is precisely the size of a minimum dominating set of  $G$ .  $\square$

To prove the running time of Algorithm 2 we will need the values  $M$  to be all smaller or equal to the depth of  $T$ . Thus we first proof the space upper bound. In the following we treat the associative arrays  $M$ ,  $M_1$  and  $M_2$  as if the entries where values between 0 and  $n$ . We will show that we can represent all values as an offset  $\leq t$  of a single value between 0 and  $n$ .

**Lemma 4.7.** *Algorithm 2 uses  $O(2^t t \log t + t \log n)$  bits of space.*

*Proof.* Let  $t$  be the depth of the provided treedepth decomposition. It is clear that the depth of the recursion is at most  $t$ . Any call to the function keeps a constant number of associative arrays and nodes of the graph in memory. By construction these associative arrays have at most  $2^t$  entries. For any of the computed arrays  $M$  the value of  $M[\emptyset]$  and  $M[S]$  for any  $S \neq \emptyset$  can only differ by at most  $t$ . We can thus represent every entry for such a set  $S$  as an offset from  $M[\emptyset]$  and use  $O(2^t \log t + \log n)$  space for the tables. This together with the bound on the recursion depth gives the bound  $O(2^t t \log t + t \log n)$ .  $\square$

**Lemma 4.8.** *Algorithm 2 runs in time  $O(3^t \log t \cdot n)$ .*

*Proof.* On a call on which  $d$  nodes of  $P_x$  are in the dominating set the associative arrays have at most  $2^s$  entries for  $s = t - d$ . As shown above the entries in the arrays are  $\leq s$  (except one). Hence, we can use fast subset convolution to merge the arrays in time  $O(2^s \log s)$  [2]. It follows that the total running time is bounded by  $O(n \cdot \sum_{i=0}^t \binom{t}{i} \cdot 2^{t-i} \log(t-i)) = O(3^t \log t \cdot n)$ .  $\square$

## 5 Conclusion and future work

We have shown that single-pass dynamic programming on treedepth, tree or path decompositions without preprocessing of the input must use space exponential in the width/depth, confirming a common suspicion and proving it rigorously for the first time. This complements previous SETH-based arguments about the running time of arbitrary algorithms on low treewidth graphs. We further demonstrate that treedepth allows non-DP algorithms that use only polynomial space in the depth of the provided decomposition. Both our lower bounds and the provided algorithm for DOMINATING SET appear as if they could be special cases of a general theory to be developed in future work and we further ask whether our result can be extended to less stringent definitions of ‘dynamic programming algorithms’.

Despite the less-than-ideal theoretical bounds of the presented DOMINATING SET algorithms, the opportunities for heuristic improvements are not to be slighted. Take the pure branching algorithm presented in Section 4. During the branching procedure, we generate all partitions from the root-path starting at the current vertex. However, we actually only have to partition those vertices that are not dominated yet (by virtue of being themselves in the dominating set or being dominated by another vertex on the root-path). A sensible heuristic as to which branch—including the current vertex in the dominating set or not—to explore first, together with a *branch & bound* routine should keep us from generating partitions of very large sets. A similar logic applies to the mixed dynamic programming/branching algorithm since the tables only have to contain information about sets that are not yet dominated. The tables could thus be kept a lot smaller than their theoretical bounds indicate.

Furthermore, it seems reasonable that in practical settings, the nodes near the root of treedepth decompositions are more likely to be part of a minimal dominating set. If this is true, computing a treedepth decomposition would serve as a form of smart preprocessing for the branching, a rough ‘plan of attack’, if you will. How much such a *guided branching* improves upon known branching algorithms in practice is an interesting avenue for further research.

It is still an open question whether DOMINATING SET can be solved in time  $(3-\epsilon)^t \cdot \text{poly}(n)$  when parameterized by treedepth. Our lower bound result implies that if such an algorithm exists, it cannot be purely dynamic programming.

**Acknowledgements** Research funded by DFG-Project RO 927/13-1 “Pragmatic Parameterized Algorithms”. Li-Hsuan Chen was supported in part by MOST-DAAD Sandwich Program under grant no. 103-2911-I-194-506.

## References

- [1] U. Bertele and F. Brioschi. On non-serial dynamic programming. *J. Combin. Theory Ser. A*, 14(2):137–148, 1973.
- [2] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets möbius: Fast subset convolution. In *Proc. of 39th STOC*, pages 67–74, 2007.
- [3] H. Bodlaender. *Dynamic programming on graphs with bounded treewidth*. Springer, 1988.

- [4] H. Bodlaender. *Treewidth: Algorithmic techniques and results*. Springer, 1997.
- [5] H. Bodlaender. Fixed-parameter tractability of treewidth and pathwidth. In *The Multivariate Algorithmic Revolution and Beyond*, pages 196–227. Springer, 2012.
- [6] H. Bodlaender, J. Deogun, K. Jansen, T. Kloks, D. Kratsch, H. Müller, and Z. Tuza. Rankings of graphs. *SIAM J. Discrete Math.*, 11(1):168–181, 1998.
- [7] R. Borie, R. Parker, and C. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(1-6):555–581, 1992.
- [8] R. Borie, R. Parker, and C. Tovey. Solving problems on recursively constructed graphs. *ACM Comput. Surv.*, 41(1):4, 2008.
- [9] E. D. Demaine, F. Reidl, P. Rossmanith, F. Sanchez Villaamil, S. Sikdar, and B. D. Sullivan. Structural sparsity of complex networks: Random graph models and linear algorithms. *CoRR*, abs/1406.2587, 2014.
- [10] R. Diestel. *Graph Theory*. Springer, Heidelberg, 4th edition, 2010.
- [11] A. Drucker, J. Nederlof, and R. Santhanam. Algorithmic paradigms through the parameterized lens. Presented at TACO Day 2016 in Aachen / personal communication.
- [12] M. Fürer and H. Yu. Space saving by dynamic algebraization. In *Lect. Notes Comput. Sc.*, pages 375–388. Springer, 2014.
- [13] G. Gutin, M. Jones, and M. Wahlström. Structural parameterizations of the mixed chinese postman problem. *CoRR*, abs/1410.5191, 2014.
- [14] M. Katchalski, W. McCuaig, and S. Seager. Ordered colourings. *Discrete Math.*, 142(1-3):141–154, 1995.
- [15] S. Lindell. A logspace algorithm for tree canonization. In *Proc. of 24th STOC*, pages 400–404, 1992.
- [16] D. Lokshtanov, D. Marx, and S. Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In *Proc. of 22nd SODA*, pages 777–789, 2011.
- [17] D. Lokshtanov and J. Nederlof. Saving space by algebraization. In *Proc. of 42nd STOC*, pages 321–330. ACM, 2010.
- [18] J. Nešetřil and P. Ossona de Mendez. *Sparsity: Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and Combinatorics*. Springer, 2012.
- [19] Michał Pilipczuk and Marcin Wrochna. On space efficiency of algorithms working on structural decompositions of graphs. In Nicolas Ollinger and Heribert Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.



- [20] J. Van Rooij, H. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Proc. of 17th ESA*, number 5193 in LNCS, pages 566–577. Springer, 2009.
- [21] P. Scheffler. *Dynamic programming algorithms for tree-decomposition problems*. Akad. d. Wissensch. d. DDR. Institut für Mathematik, 1986.
- [22] R. Schreiber. A new implementation of sparse gaussian elimination. *ACM Trans. Math. Software*, 8(3):256–276, 1982.